Usage of Functions

Functions, as do subprograms, give you the possibility to receive data, to change it and to give the results to the calling module. The advantage of using functions over subprograms is that function calls can be used directly in statements and expressions without the need for additional temporary variables.

Normally, depending on the parameters that are given to the function, the result is produced in the function and is returned to the calling object. If other values are to be returned to the calling module, this can be done by using the parameters (see the section Subprogram).

Once the function code has been completely executed, control is given back to the calling object and the program continues with the statement that comes after the function call.

For more information about the Natural object type "Function", see the section Object Types in the Natural Programming Guide.

Advantage of Function Calls

The following two examples show the difference between using function calls and subprograms.

Example using Function Calls:

Program Object:

```
/* Natural program using a function call
INCLUDE C#ADD
WRITE #ADD(< 2,3 >) /* function call; no temporary variable necessary
END
```

Function Object:

```
/* Natural function definition
DEFINE FUNCTION #ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #ADD := #SUMMAND1 + #SUMMAND2
END-FUNCTION
END
```

Copycode Object (e.g.: "C#ADD"):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```


If you want to achieve the same functionality by using a subprogram, you must use temporary variables.

Example using a Subprogram:

Program Object:

```
/* Natural program using a subprogram
DEFINE DATA LOCAL
1 #RESULT (I4) INIT <0>           /* temporary variable
END-DEFINE

CALLNAT 'N#ADD' USING #RESULT 2 3 /* result is stored into #RESULT
WRITE #RESULT                     /* print out the result of the subprogram
END
```

Subprogram Object (e.g.: "N#ADD"):

```
/* Natural program using a subprogram
DEFINE DATA PARAMETER
1 #RETURN   (I4) BY VALUE RESULT
1 #SUMMAND1 (I4) BY VALUE
1 #SUMMAND2 (I4) BY VALUE
END-DEFINE

#RETURN := #SUMMAND1 + #SUMMAND2
END
```

Function Definition

The function definition contains the Natural code to be executed when the function is called. As with subprograms, you need to create an object of type "Function" which contains the function definition.

The function call itself can be in any object type which contains executable code. It cannot appear in class objects, for example.

To be able to compile function calls, Natural needs information about the type of the return value. This information is then made available to the compiler in the prototype definition. You can also include the definition of the parameter to be passed back, which is then checked at compile time.

Since Natural makes the connection between "calling" and "called" objects at runtime, and not before, the computer does not know the type of a function return value it is dealing with at compile time. This is due to the fact that the object containing the function does not necessarily have to exist [at compile time]. It is for this reason that the prototype definition was created, so that the datatype can be generated into the generated program at compile time.

It is important to remember that a prototype definition never contains executable code. A prototype definition simply contains the following information about the function call: the type of the return value or the parameter being passed back.

Symbolic and Variable Function Call

See the section Function Call for more details about this topic.

Prototype Cast

In order to find the corresponding prototype of a specific function, a prototype is searched for which bears the name of the function. If this is not the case, it is assumed that the function call is symbolic. In this case, the function "signature" must be defined by using the keyword "PT=" in the function call.

Recursive Function Call

If a function is to be called recursively, the function prototype must be contained in the function definition, or be inserted by means of an INCLUDE file.

Example:

Function Object

```
/* Function definition for calculation of the math. factorial
DEFINE FUNCTION #FACT
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  LOCAL
  1 #TEMP (I4)
  END-DEFINE

  /* Prototype definition is necessary
  INCLUDE C#FACT

  /* Program code
  IF #PARA=0
    #FACT := 1
  ELSE
    #TEMP := #PARA - 1
    #FACT := #PARA * #FACT(< #TEMP >)
  END-IF

END-FUNCTION
END
```

Copycode Object (e.g.: named "C#FACT"):

```
/* Prototype definition is necessary
DEFINE PROTOTYPE #FACT
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

Program Object:

```
/* Prototype definition
INCLUDE C#FACT

/* function call
WRITE #FACT(<12>)
END
```


Behavior of Functions in Statements and Expressions

Function calls can be used directly in statements or expressions, replacing operands. However, this is only allowed in places where operands cannot be modified.

All function calls are executed according to their syntactical sequence which is analyzed at compile time. The results of the function calls are saved in internal temporary variables and passed onto the statement or expression.

This fixed sequence makes it possible to allow and execute standard output in functions, without, for example, unwillingly influencing the output of a statement.

Example:

Program:

```
/* Natural program using a function call
INCLUDE CPRINT
PRINT 'before' #PRINT(<>) 'after'
END
```

Function Object:

```
/* Natural function definition
/* function returns integer value 10
DEFINE FUNCTION #PRINT
  RETURNS (I4)
  WRITE '#PRINT'
  #PRINT := 10
END-FUNCTION
END
```

Copycode (e.g.: "CPRINT"):

```
DEFINE PROTOTYPE #PRINT END-PROTOTYPE
```

The following is the result which is then sent to the standard output:

```
#PRINT
before       10 after
```

Usage of Functions as a Statement

Functions can also be called as statements independently from statements and expressions. In this case, the return value-assuming it has been defined- is not taken into account.

If, however, an independent function is declared after an optional operand list, the operand list must be followed by a semicolon to make it clear that the function call is not a part of the operand list.

Example

Program Object:


```
/* Natural program using a function call
DEFINE DATA LOCAL
1 #A (I4) INIT <1>
1 #B (I4) INIT <2>
END-DEFINE

INCLUDE CPROTO

WRITE #A #B
#PRINT_ADD(< 2,3 >) /* function call belongs to operand list just in front of it

WRITE '******'

WRITE #A #B;        /* semicolon separates operand list and function call
#PRINT_ADD(< 2,3 >) /* function call doesn't belong to the operand list
END
```

Function Object:

```
/* Natural function definition
DEFINE FUNCTION #PRINT_ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #PRINT_ADD := #SUMMAND1 + #SUMMAND2
  PRINT '#PRINT_ADD =' #PRINT_ADD
END-FUNCTION
END
```

Copycode Object (e.g.: named "CPROTO"):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #PRINT_ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

